LightWSP: Whole-System Persistence on the Cheap

Yuchen Zhou
Purdue University
West Lafayette, USA
zhou1166@purdue.edu

Jianping Zeng
Purdue University
West Lafayette, USA
zeng207@purdue.edu

Changhee Jung
Purdue Univerisity
West Lafayette, USA
chjung@purdue.edu

Abstract*—Whole-system persistence (WSP) has recently attracted more interest thanks to its transparency and performance benefits over partial-system persistence where users are not only burdened by complex persistent programming but also incapable of using DRAM as LLC. Nevertheless, existing WSP work either introduces high hardware cost or causes non-trivial performance overhead. To this end, this paper presents LightWSP, a compiler/architecture co-design scheme that can achieve WSP in a lightweight yet performant manner. LightWSP compiler partitions program into a series of recoverable regions (epochs) with their live-out registers checkpointed, while LightWSP hardware persists the stores of the regions—whose boundary serves as a power failure recovery point—enforcing crash consistency; LightWSP leverages the battery-backed write pending queue (WPQ) of a memory controller as a redo buffer, *i.e.,* all stores are first buffered in WPQ and then persisted together in non-volatile memory (NVM) at each region end. In this way, no matter when power failure happens, NVM is never corrupted by the stores of the power-interrupted region, facilitating correct recovery. In particular, LightWSP supports multiple memory controllers on the cheap without costly speculation/misspeculation handling mechanisms used by prior work. The experimental results with 38 applications show that LightWSP incurs only an average of 9.0% run-time overhead. This is on par with the state-of-the-art work, that complicates the core microarchitecture significantly with its intrusive design for memory controller speculation, yet the hardware cost of LightWSP is near zero (0.5B per core).

I. Introduction

The emergence of non-volatile memory (NVM) technologies, exemplified by Intel's Optane Persistent Memory [42], has paved a new way for data persistence thanks to their irresistible benefits, *e.g.,* nonvolatility, high areal density, DRAM-comparable speed, and byte-addressability[1] [1], [2], [56], [61]. The upshot is that the core pipeline can directly persist the data being stored to persistent memory (PM) with regular store instructions. This allows program to achieve high-performance data persistence without having to rely on slow block devices.

However, care must be taken to avoid crash inconsistency across power failure. During program execution, a younger store could be evicted (persisted) from cache to NVM before an older one; if power failure occurs while the older is yet to be persisted, it can lead to inconsistent NVM states in the wake of the power failure. With that in mind, users typically leverage transactions [20], [51], [59], [65], [95], [113] or failure-atomic-sections (FASEs) [13], [21], [38], [48], [50], [66], [96]—atop Intel Optane's *app-direct mode* [45] where NVM is used as persistent heap—to delineate a specific code region and persist its data with crash consistency ensured. This is so-called partial-system persistence (PSP) in that only the stores of such user-defined code regions are to be persisted in NVM.

While PSP looks promising for achieving data persistence and crash consistency, it faces several challenges. First, to ensure persistence, PSP either uses expensive persist barriers—*e.g.,* `clwb` and `sfence` in x86 processors—or relies on dedicated hardware structures [51], [60], [80], [96], thus being unable to achieve low-cost data persistence. Second, in PSP, application-specific recovery code—tailored to data structures being persisted—is often desired for performance reasons, which places a significant programming burden on users [7], [37], [91], [93], [111]. Third, PSP requires users to explicitly manipulate the persistent heap with special memory allocator and deallocator such as `pmalloc/pfree` [44], exposing them to potential memory leaks. This further complicates persistent programming and increases the risk of bugs [14], [25], [29], [32]–[34], [58], [71]–[73], [76], [77], [82].

More importantly, it is quite challenging for PSP to exploit DRAM as last-level cache (LLC) atop non-volatile main memory. The reason is twofold: (1) existing technologies, *e.g.,* Intel eADR [41], cannot afford to keep volatile states of terabyte DRAM from power failure; (2) DRAM is still used as main memory in PSP—unlike Optane's *memory mode* that allows DRAM cache yet uses PM as *volatile* main memory losing persistence. Consequently, PSP forfeits the benefit of leveraging DRAM as LLC, though it is essential for high performance; an ideal PSP scheme underperforms LightWSP, that enables DRAM cache, by 46.8% on average for memory-intensive applications and up to 160% for `libquantum` of CPU2006 as shown in Figure 9.

As an alternative to PSP, whole-system persistence (WSP) has recently attracted growing interest among researchers [15], [19], [27], [53], [81], [98], [105], [108], [110]. Two unique features of WSP drive this trend, *i.e.,* transparency and performance. WSP ensures persistence for all data of every kind of program—including OS and libraries—with no source code modification. Moreover, WSP can enable DRAM cache over PM by using it as non-volatile main memory without compromising data persistence or crash consistency. Nevertheless, existing WSP work is far from practical due to intrusive hardware modification and/or high run-time overhead. For example, Capri [53] resorts to over-complex redo+undo log-

[1]NVM DIMMs are attached to the regular memory bus as with DRAM DIMMs.

2379-315/24/$31.00 ©2024 IEEE
DOI 10.1109/MICRO61859.2024.00025

ging techniques for correct power failure recovery, causing significant performance overhead—especially in the presence of multiple memory controllers (MCs). While PPA [108] achieves high performance with its simple recovery that replays unpersisted stores in the wake of power failure, it requires store operand registers to be intact, which is enforced in the register renaming stage of the pipeline and thus increases the critical path, giving significant pressure on the clock cycle optimization. Recently, cWSP [110], the state-of-the-art, takes a different recovery approach based on forming *idempotent* regions [23], [66]–[69], [112] whose re-execution(s) across power failure should produce the same designated output. Despite the software-based idempotent recovery, cWSP ends up changing both core microarchitecture and MCs to realize its *memory controller speculation* that not only incurs recurring communications between cores and MCs to track the persistence status of each region but also requires special undo logging hardware for misspeculation handling.

With the aforementioned in mind, this paper proposes LightWSP, a compiler/architecture co-design scheme that can achieve WSP in a lightweight yet performant manner. LightWSP repurposes existing hardware components leaving the core microarchitecture as is, *e.g.,* Intel's non-temporal data path [47], [87] as a FIFO persist path—with the write-combining buffer [46] disabled—and its battery-backed write pending queue (WPQ) [43] as a redo buffer. To correctly recover from power failure, LightWSP compiler partitions program into a series of recoverable regions (epochs) with their live-out registers checkpointed, while LightWSP hardware persists the stores of the regions, whose boundary serves as a power failure recovery point, enforcing crash consistency. That is, all stores are first buffered in WPQ and then persisted in PM at each region end as with *buffered epoch persistency* [54], [80]. Therefore, no matter when power is cut off, PM is protected against the stores of the power-interrupted region as if it were not started, which facilitates correct power failure recovery.

Although the *region-level persistence* requires persist ordering between regions (epochs), naive use of `sfence` at each region boundary causes significant performance overhead. To this end, LightWSP proposes a novel technique called *lazy region-level persist ordering* that allows cores to execute a region as soon as its preceding one finishes, by delegating the persist ordering of regions to MCs; it is *lazy* in that the ordering happens at the very end of the persist path, unlike cWSP involving cores at the front in tracking how regions persist. In particular, the *lazy region-level persist ordering* works with no hassle even for multiple MCs; LightWSP simply ensures that each MC is aware of the execution order of regions and holds their stores until all of them arrive in the WPQ to persist them all together. That way, even if the stores of a younger region reach the WPQ—due to MC's non-uniform memory access—outrunning older regions, none of the stores can ever persist before any other stores of older regions do. The takeaway is that LightWSP can achieve high performance for server-class manycore systems backed with multiple MCs at a low cost.

The evaluation results with a total of 38 applications from SPEC CPU2006/2017, SPLASH3, STAMP, NPB, and WHISPER show that LightWSP only incurs a 9.0% run-time overhead on average and near zero hardware overhead, making LightWSP highly suitable for the implementation in real silicon. The contributions of LightWSP are summarized below:

- Ultra lightweight solution to practical whole-system persistence, with only 0.5B per core hardware cost.
- *Lazy region-level persist ordering* that completely hides persistence latency, allowing LightWSP to achieve high performance, *i.e.,* 9.0% average run-time overhead.
- High potential to work for the future CXL technology as shown in Section V-F6.

II. Background and Motivation

A. Non-Temporal Data Path

Intel classifies program data into two categories: temporal data, expected to be reused, and non-temporal data, used once and not likely to be reused shortly [47], [87]. To optimize cache efficiency, it is advantageous to store only temporal data in caches while not keeping non-temporal data there; caching non-temporal data rather degrades the utilization of the caches by occupying their space in vain, *i.e.,* cache pollution. To mitigate this issue, Intel processors are equipped with a non-temporal data path that bypasses the entire cache hierarchy and directly conveys the data from the store buffer to memory via `nt-store`. Even though the non-temporal data path was initially designed to minimize the cache pollution, prior work [92] repurposed it—with its write combining buffer disabled—as a persist path that we believe can be used to achieve whole-system persistence [110].

B. Crash Inconsistency in Multiple Memory Controllers

While the non-temporal data path supports the FIFO order of data placed therein, their final persistence can be out of order in the presence of multiple memory controllers (MCs) due to the non-uniform memory access (NUMA) effect. Specifically, the distance between each core and the same MC is different from each other, so the resulting core-to-PM latency varies depending on where to persist. This is problematic because LightWSP's region-level (epoch) persistency requires—for crash consistency—persisting regions in order, *i.e.,* stores should be strictly ordered across a region boundary—though they can be reordered within each region (epoch). Unfortunately, stores of a younger region may persist before those of the older, provided the former target a near MC while the latter do a far MC. To address this issue, prior work either waits at each region end before moving on to the next region, which incurs significant performance overhead [22], [31], [51], [60], [80], or introduces complex hardware support to guarantee that power-interrupted program can be recovered from any inconsistent states [110].

C. Whole-System Persistence (WSP)

To achieve data persistence and crash consistency while enabling DRAM cache (LLC) atop PM (main memory), whole-system persistence WSP could be the magic bullet. Recent studies have explored techniques for WSP, and they can be categorized as JIT-checkpoint-oriented and non-JIT-checkpoint-oriented approaches[2] to a large extent.

1) JIT-Checkpoint-Oriented Approach: This kind of design takes advantage of a voltage monitor to detect impending power failure that triggers necessary state saving right before the outage—thus being called just-in-time (JIT) checkpointing. For example, Narayanan et al. proposed to persist all volatile data when power is about to be cut off—using the energy remaining in a power supply unit (PSU) [81]. However, an excessive amount of residual energy needs to be secured to persist the deep cache hierarchy of modern manycore processors [6], which gets worse for the off-chip DRAM cache as in Intel Optane's memory mode. According to LightPC [63], another work that also exploits the residual energy to persist volatile data, the server-class PSU can persist at most 64 cores with 40MB cache, while standard ATX PSU can only do at most 32 cores with 16KB cache. This implies that it is impossible to persist the huge DRAM of typical servers with the residual energy of PSU, *i.e.,* simply JIT checkpointing all volatile states upon power failure cannot achieve WSP at a low cost.

2) Non-JIT-Checkpoint-Oriented Approach: With that in mind, Jeong et al. propose Capri [53] that leverages a separate data path connecting L1 cache to PM, *i.e.,* data reaching the cache should be persisted in PM through the persist path. This allows Capri to avoid checkpointing all the volatile data including the DRAM cache in the event of power failures. For their correct recovery, Capri guarantees that all its data are persisted in a region granularity and provides its regions with hardware-based failure atomicity.

However, there are several problems making it hard to deploy Capri in practice. First, Capri puts a huge bandwidth pressure on its separate persist path, since every 8B store generates a 64B cacheline flush—though the rest 56B remains the same. Second, Capri incurs substantial hardware cost. To guarantee failure-atomic region persistence, Capri introduces two hardware queues per core, *i.e.,* front-end and back-end buffers whose entry maintains both undo and redo logs in addition to data being stored, causing non-trivial hardware cost (54KB per core). Last but not least, it is not clear how Capri can be extended to support multiple MCs—that are commonly found in today's HPC and cloud servers—without complicating its already expensive hardware support.

In light of this, PPA [108] puts forward a simpler design that avoids complex logging mechanisms. The key idea is that power-interrupted program can be correctly recovered by replaying unpersisted stores when power comes back. To realize this, PPA (1) enforces *store integrity*, *i.e.,* the operand registers of stores must be held in a physical register file (PRF) even after they are committed, and (2) JIT checkpoints the PRF—including their registers and PC—upon power failure. Whenever PRF runs out being incapable of enforcing store integrity further, which serves as an implicit region (epoch) boundary, PPA lets all pending stores persisted so that their registers are reclaimed to resume the store integrity for the next region. In the wake of power failure, PPA can thus replay the stores of the power-interrupted region with their registers restored and then resume from the interruption point (through the checkpointed PC) to achieve correct recovery. An an optimization, PPA tries to hide the latency of the store persistence at each region boundary by eagerly writing back every store as soon as it reaches L1 cache. That way, its persistence can be overlapped with the execution of subsequent instructions in the same region.

However, the effectiveness of the optimization hinges on the length of a region, which is heavily affected by the PRF utilization. For example, when program demands a lot of physical registers in some phase, the resulting region would not be long enough to have a sufficient amount of computation therein for the complete hiding of the persistence latency. More importantly, enforcing PPA's store integrity turns out to be expensive in terms of hardware cost. That is because PPA introduces extra complexity to the critical path of the pipeline, *e.g.,* its renaming stage must ensure that the mapped physical registers are not held for store integrity, thereby putting significant pressure on the CPU's clock cycle optimization. Such an intrusive pipeline modification may hinder the wide adoption of PPA in industry.

The state-of-the-art, cWSP [110], pursues a software-based approach, seeking to minimize the cost of necessary hardware. It leverages idempotent region formation for correct power failure recovery [23], [66]–[69], [112]. The cWSP compiler divides program into a sequence of idempotent regions—with stores persisted in a region granularity—so that the re-execution(s) of a power-interrupted region can produce the same designated output. Thanks to the side-effect-free nature of an idempotent region, cWSP can recover from power outage(s) occurred in the middle of the region by simply restarting it. To achieve high-performance crash consistency in the presence of multiple MCs, cWSP never ensures in-order persistence despite their NUMA effect. That is, before the stores of a region complete their persistence, cWSP speculatively persists the following regions' stores with their data undo logged guessing that the resulting out-of-order persistence is not caught by power failure; this is so-called memory controller speculation. In case of misspeculation, *i.e.,* power failure, cWSP can revert the inconsistent PM states by consulting the undo logs and resume from the end of the most recently persisted (non-speculative) region for recovery.

Unfortunately, maintaining the speculation status of regions, *i.e.,* identifying which regions are speculative and which are not, makes cWSP's core and MCs continuously communicate with each other to keep track of whether each region has completed the persistence of all its stores. This leads to

[2]Although they may also use JIT checkpointing to save some necessary status, it is not the main idea that characterizes their proposed techniques.

intricate hardware modification for both cWSP's core microarchitecture and its MCs. In particular, the MCs need additional hardware support to reduce the delay of each PM write which is caused by the undo logging—that must copy the original data before the write. Since this delay is imposed on virtually all stores (though it is mitigated), cWSP may cause significant performance degradation especially for write-intensive workloads. Given all this, there is a compelling need for lightweight yet performant WSP that can be practically realized at a low cost.

III. LightWSP Approach

The goal of LightWSP is providing a lightweight solution to whole-system persistence (WSP) without compromising on performance. Figure 1 shows the architecture overview of LightWSP. It repurposes the non-temporal path for data persistence and exploits compiler/architecture interaction for low-cost power failure recovery; the compiler generates recoverable regions, while the architecture is responsible for their crash-consistent execution (§III-A, §III-C). As shown in the figure, LightWSP leverages MC's battery-backed write pending queue (WPQ) as a redo buffer, *i.e.,* each region's stores are first buffered in their target WPQ so that they later persist together failure-atomically. For high-performance data persistence, LightWSP proposes *lazy region-level persist ordering (LRPO)* that allows cores to execute regions without persistence delay across boundary, while enforcing their persist order in the background (§III-B). In particular, LightWSP guarantees correct persist order even for multi-threaded program with no hassle (§III-D) thanks to its simple LRPO and recovery mechanisms (§III-E). The rest of this section gives the overview of LightWSP's design components.

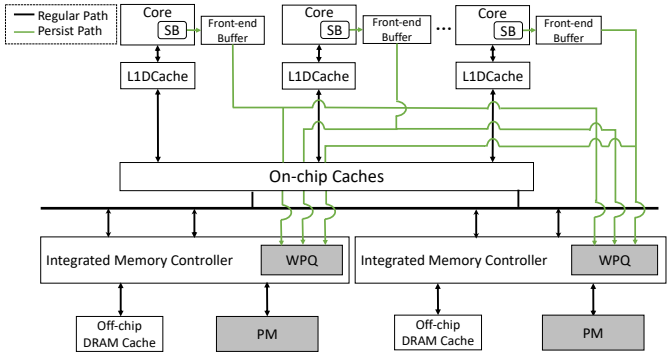

Fig. 1. LightWSP overall architecture; grey signifies non-volatile parts.

A. Region-Level Persistence

To avoid expensive JIT checkpointing, LightWSP exploits the non-temporal path[3] for data persistence in an 8B granularity (*c.f.,* 64B for Capri). This implies that LightWSP cannot resume program from the point of power failure on which JIT-checkpoint-oriented approaches (§II-C1) persist PC to pick up thereafter (*i.e.,* roll-forward recovery). Instead, LightWSP resorts to roll-back recovery using region-level persistence [4], [16]–[18], [114] where power failure in a middle of a region can be recovered by directing the PC back to the beginning of the unpersisted region in the wake of the failure. For this purpose, LightWSP compiler partitions program into a series of recoverable regions with live-out registers checkpointed via store instructions. In particular, LightWSP regards a region as **persisted** only after all its stores reach their target MCs. To equip each region with failure atomicity, LightWSP should ensure that all stores of the region persist in an all-or-nothing fashion.

LightWSP realizes this by (1) tagging stores with their region ID, (2) broadcasting the ID at each region end (*i.e.,* boundary) to MCs for them to be aware of (un)persisted regions, and (3) gating the battery-backed WPQ of MCs which guarantees to flush only the entries of persisted regions to PM upon power failure. That is, WPQ quarantines (gates) the stores of each region until it is persisted, in which case its quarantined stores can finally commit to PM in a bulk manner. If power failure occurs before the region ID is broadcast to MCs, *i.e.,* the region has not reached the end, its every store quarantined in their WPQs are discarded without affecting the PM states. In this way, LightWSP can correctly recover from power outages no matter when they occur in that PM always remains consistent—which serves as a basis for its crash consistency.

B. Lazy Region-Level Persist Ordering

While the *region-level persistence* requires persist ordering between regions for correct recovery, naive use of `sfence` at each region boundary causes significant performance overhead. To this end, LightWSP proposes *lazy region-level persist ordering* (LRPO) that allows cores to start their region without waiting for the prior region to persist by offloading the enforcement of regions' persist order to MCs. With the help of LRPO, the persistence latency of a region can be overlapped with the execution of the following region(s), as shown in Figure 2. Since the ordering is enforced at the end of the persist path, *i.e.,* MCs, it is called *lazy* region-level persist ordering.

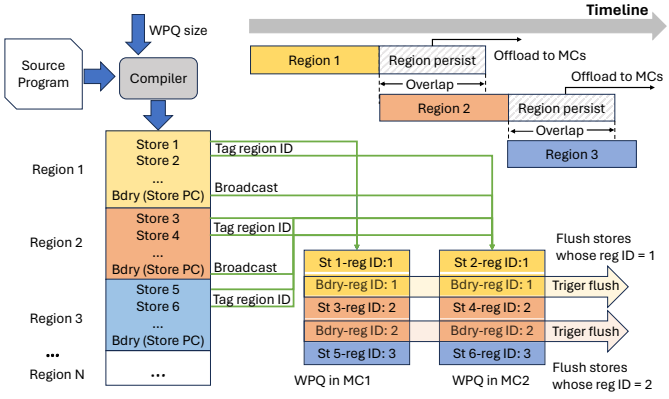

Fig. 2. Lazy region-level persist ordering.

The beauty of LRPO is that it just works fine for multiple MCs, unlike cWSP complicating its base design to

[3]LightWSP repurposes Intel's write-combining buffer (WCB) as the front-end buffer alongside the store buffer to release its stores as soon as possible.

support them; more details are deferred to §IV-B. Note that LightWSP's LRPO is fundamentally different from PPA's *region-level asynchronous persistence*. To clarify, PPA's store persistence starts right after the stores are merged to L1 cache, performing in parallel with the subsequent instructions of the same region, not the following region(s). In other words, PPA can only hide the persistence latency with its in-region ILP—whereas LightWSP can with cross-region ILP that overlaps the persistence latency of a region with the execution of those instructions from many following regions. At each hardware-delineated region boundary of PPA, if any of preceding stores has not finished its persistence yet, the pipeline must stall until they are all persisted.

C. Region Partitioning

There are two main goals of the region partitioning: (1) guaranteeing the failure atomicity of region-level persistence and (2) mitigating core pipeline stalls under LRPO. First, for the failure atomicity, the store count of each region must not exceed the size of WPQ to prevent its overflow to keep PM states intact. Otherwise, they could be corrupted by the stores of any power-interrupted region which are spilled to PM. Second, since LRPO lets cores push stores to WPQs—via persist paths—with no delay across regions, WPQs' flush rate might not keep up with cores' push rate. In this case, the WPQs quickly fill up and thus exert back pressure on the core, thereby stalling the pipeline. To mitigate this issue, LightWSP compiler limits the region size. This is because a shorter region takes less time to be flushed and leaves more space for WPQs, which allows them to accommodate stores from consecutive regions without stalls.

With the aforementioned in mind, LightWSP puts a cap on how many stores can ever exist in each region. Based on empirical analysis, LightWSP compiler sets such an in-region store count threshold to half of the WPQ size (see §IV-A for details). As a result, WPQ can accommodate at least two regions' stores without waiting for the persistence latency, *i.e.,* each region can have its stores inserted to WPQ before the prior region's stores are all committed to PM. Nonetheless, it turns out that even with the half-of-WPQ threshold, WPQ can handle stores from more than two regions most of the time. The reason is two-fold: (1) the number of stores in each region is usually smaller than the threshold (§V-G3); (2) WPQ starts to flush the stores of a **persisted** region immediately after receiving its region ID (boundary shown as right arrows in Figure 2), giving room for the second next region.

In particular, there is a special case where the guarantee of zero WPQ overflow needs to be relaxed for deadlock avoidance. For instance, under multi-threading (or processing), WPQ might be fully occupied by stores of many regions with none of the regions being persisted, which leads to a deadlock. That is because WPQ here waits to receive the region IDs (boundaries) to make room with flushing, while it cannot accommodate any of them due to the lack of room. However, this case turns out to be very rare; §IV-D shows how LightWSP handles the case.

D. Ensuring Persist Order in Multi-Threaded Program

The persist paths of different cores operate concurrently, and their stores may arrive at MCs in an arbitrary order. Thus, care must to taken to enforce the inter-thread persist order for multi-threaded program with conflicting data accesses. For example, in the presence of inter-thread dependencies, *e.g.,* WAW (write-after-write) and RAW (read-after-write), the persist order may be disrupted unless the memory access order is preserved. However, unlike the prior work [31], [60], [80], LightWSP cannot simply preserve the order by leveraging cache coherence in that its persist paths bypass caches.

To this end, LightWSP takes advantage of the unique characteristic of data-race-free (DRF) program[4], *i.e.,* the conflicted memory accesses of concurrent threads must be explicitly managed in program by a synchronization primitive to order them strictly. For instance, programmers should protect such memory accesses using locks/semaphores or custom signal-wait mechanisms properly implemented for the underlying memory consistency model.

In order to follow the original happens-before order that is dynamically established by synchronization primitives, LightWSP compiler inserts region boundaries when encountering their instructions, *e.g.,* memory fences or atomic operations. This allows the happens-before order to be derived by the region execution order that is conveyed MCs—to let LRPO comply with the order for its persistence (see more in §IV-C).

E. Failure Recovery Protocol

With the help of LightWSP's region-level persistence (§III-A), it is quite simple to recover from power failure no matter when it occurs. Upon power failure which is detected by the voltage monitor, MCs check the region persistence status by leveraging a region ID so that their WPQ can flush only those entries from persisted—thus keeping the PM states consistent; all other entries in the WPQ naturally disappear with the power failure. When power comes back, LightWSP lets each core resume the power-interrupted program from the beginning of the latest unpersisted region, *i.e.,* the end of the most recently persisted region, for correct recovery (see §IV-F for more details).

IV. Implementation Details

A. LightWSP's Compiler

Threshold Determination: As a criterion for the region formation (§III-C), LightWSP compiler requires a threshold of store count per region which technically limits how many stores can exist in each region. To suppress the WPQ overflow, our threshold is empirically set as half of the WPQ size so that the WPQ can accommodate stores from at least two regions without stalls. At first glance, one might want to pick a threshold a lot smaller than the half to reduce the WPQ pressure for fewer stalls; however, this results in many tiny regions and in turn increases the number of checkpoint

[4]Existing persistent programming models already assume DRF program as their target [13], [38], [48], [66]

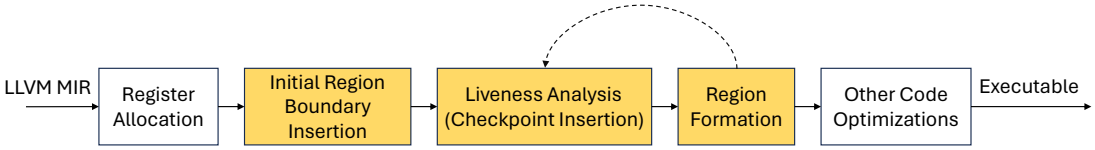

Fig. 3. The workflow of LightWSP compiler; yellow indicates newly added LLVM passes.

stores for saving their live-out registers, thereby leading to potential performance degradation. Despite, a large threshold is not necessarily better than the small one; compared to a small threshold, the WPQ under a large one quarantines more stores per region for a longer time, which effectively increases the WPQ pressure and possibly makes the following region(s) wait for room in the WPQ. According to our experiment for exploring the threshold, setting it as half of WPQ can achieve high region-level persistence efficiency, while reducing the number of necessary checkpoint stores to the most. In fact, our threshold accomplishes virtually no structural hazard on WPQ (§V-C), and therefore it rarely exerts back pressure on the core pipeline via the persist path. That is, LightWSP's region-level persistence incurs near-zero pipeline stall!

Overall Workflow: LightWSP compiler performs region partitioning with WPQ size in mind as inspired by previous work [4], [16]–[18], [106], [114]. This optimization is conducted at the LLVM MIR level and is scheduled after register allocation. The overall workflow is depicted in Figure 3. To clarify, during the region partitioning process, the compiler counts the number of stores while traversing the control flow graph (CFG) of the program. When the store number reaches the predefined threshold (*i.e.,* half of the WPQ size), LightWSP compiler inserts a region boundary instruction. Then, the compiler analyzes the live-out [5] registers in each region and checkpoints them to the checkpoint storage—indexed by register number—in PM. At each region end (boundary), the program counter (PC) is also stored (checkpointed) to serve as a recovery point in case the following region gets interrupted by power failure.

However, it is challenging to perform the region partitioning and the live-out register checkpointing together due to their circular dependence. On the one hand, the region partitioning pass determines where to place region boundaries and therefore affects live-out registers in each region. On the other hand, the live-out register checkpointing pass inserts more stores, which in turn affects where to place region boundaries. For this reason, LightWSP compiler takes the same strategy as prior work [17], [18], [39], [53], [57], [70], [106], [107], [109], [114] to break the circular dependence as demonstrated in the following.

Initial Region Boundary Insertion: LightWSP compiler first treats function callsites as region boundaries. Then, it inserts a region boundary at the entry and exit points of each function. In particular, to avoid the number of stores exceeding the threshold during loop execution, LightWSP compiler also places a region boundary at the header of every loop—unless it has no stores. That way, each loop iteration forms a region; of course, additional boundaries are placed in the loop body, if necessary, to ensure that no region therein has more stores than the threshold. In addition, as discussed in §III-D, the compiler also inserts region boundaries when meeting memory fences and atomic operations to guarantee correct persist order for multi-threaded program.

Checkpoint Store Insertion: With these initial region boundaries in mind, LightWSP compiler performs liveness analysis [5] over the partitioned CFG to figure out live-out registers in each region. To facilitate this, the compiler splits the basic blocks that have region boundaries therein, ensuring that regions always start at the beginning of basic blocks; this simplifies the computation of live-out registers of each region. For the resulting live-out registers of each region, LightWSP compiler inserts store instructions—to checkpoint the registers—right after their last update point in the region.

Region Formation: Then, LightWSP compiler tries to combine the initial regions with store counts below the threshold into larger ones by traversing CFG again in topological order. By doing this, the compiler can enlarge the region size and usually eliminate many checkpoint stores, since some live-out registers of a combined region are no longer live-out if the following region redefines the registers. After the region combining, the store count of the merged region may exceed the store threshold. If this occurs, the compiler places a new boundary within the region to ensure its stores never exceed the threshold and recalculates the number of live-out registers of the newly partitioned regions. This combining/repartitioning process is repeated until no region has more stores than the threshold, effectively resolving the issue of the aforementioned circular dependence. Nevertheless, it is important to note that the resulting regions do not necessarily have exactly the same number of stores as the threshold; instead, the threshold indicates the maximum number of stores allowed per region.

Region Size Extension and Checkpoint Pruning: Placing region boundaries at the beginning of each loop header can result in numerous small regions if their loop bodies contain only a few stores. This can increase checkpoint stores due to additional live-outs across that many region boundaries. To address this issue, for loops with known iteration counts, LightWSP compiler leverages *loop unrolling* to enlarge the region size. Besides, for other loops with unknown iteration counts, the compiler utilizes the *speculative loop unrolling* [39], [53] to extend the region size. The idea is that the compiler duplicates the loop body and the loop exit condition simultaneously while enforcing the store count threshold. As a result, LightWSP compiler can form almost 3x longer regions and reduce checkpoint stores by 3x as well.

In addition, to further reduce the checkpointing overhead, LightWSP compiler also leverages checkpoint pruning, *i.e.,* it removes live-out checkpoints if their register values can be reconstructed using other checkpointed values available at recovery time [15], [49], [53], [57].

Checkpoint Storage Management: Since checkpoint stores have fixed destination addresses, it allows the saved register values to be accessed via an index within an array during power failure recovery. Given this, LightWSP compiler allocates a PM-resident global array where all registers are mapped to dedicated slots, *e.g., r*0 is mapped into the index zero. This is feasible because the number of architectural registers is already defined by the ISA. In the wake of power failure, LightWSP recovery runtime gets all necessary register values ready by reloading from the checkpoint storage or reconstructing from the reloaded values in order to achieve correct recovery.

I/O Functions: Supporting irrevocable operations such as I/O operations has still remained an open problem. However, recall that LightWSP compiler places region boundaries at each function call, and therefore those functions that implement I/O operations are considered separate regions. We believe LightWSP can handle I/O operations by checkpointing the necessary status before the I/O operations start, allowing power-interrupted I/O operations to be restarted in the wake of power failure.

B. Persist Order Guarantee under Lazy Region-Level Persist Ordering

Due to LRPO and WPQ-size aware region formation, WPQ is populated with stores from several regions. However, the persist order of the regions does not necessarily match with their execution order in the presence of multiple MCs that lead to non-uniform memory access (NUMA) (§II-B). That is, the stores of a younger region could arrive at WPQ before the old one's stores. LightWSP addresses the problem by making MCs aware of the region execution order and enforcing the correct persist order. To achieve this, LightWSP (1) tags stores with their region ID and (2) broadcasts the ID at the region end. First, all stores leaving store buffers are tagged with HW-managed counter `region ID`—encoded as the first 16 bits of the address which are unused in modern OS[5]. Second, at each region end (boundary), *i.e.,* the PC-checkpointing store tagged with the region ID, it is broadcast to MCs and then atomically increases by 1. That way, for WPQ entries of the MC, it can identify which regions they belong to and infer the execution order of the regions.

On the other hand, each MC manages a `flush ID` that refers to the latest unpersisted region. It increments by 1 when a region is **committed**, *i.e.,* all of its stores are flushed to PM. Once MCs receive region ID ('bdry: Store PC' in Figure 2), it triggers the flush of the region's stores buffered in their WPQ. For this purpose, LightWSP makes the WPQ flush only those entries whose region ID matches with the flush ID; any mismatched WPQ entries—from younger region(s)—still remain until the flush ID gets updated for matching. Note that the flush ID is stored in a dedicated persistent register which is managed by controller logic in MC as with the previous work [115].

The following demonstrates a complete process of how a region is persisted to PM. When a region finishes, its region ID (boundary) is broadcast to all MCs through a NoC router[6]. If all MCs receive the boundary (*i.e.,* the region is persisted), their WPQ starts flushing the entries if matched. To make sure the boundary reaches all MCs, any MC sends bdry-ACKs to other MCs upon receiving the boundary. Once each MC obtains the ACKs from all other MCs, *i.e.,* the broadcast ID (boundary) has reached all MCs, its WPQ starts to flush the stores whose tagged region ID is the same as the flush ID of the MC. After flushing them all, the MC sends flush-ACKs to other MCs, which is necessary to update the flush ID of other MCs, in case the stores of a region are spread to all MCs. After each MC gets the ACKs from others, it recognizes that the region—whose ID matches with the flush ID—is finally committed, thus increasing its flush ID.

In particular, a couple of things worth mentioning. First, such in-flight communication of both ACKs is guaranteed to reach their targets in the event of power failure by using MC-resident battery—that is anyway necessary to keep WPQ in the persistence domain as with Intel ADR. Second, LightWSP's LRPO can naturally hide the latency of the ACKs communication—that can be built atop Intel QuickPath Interconnect (QPI) offering a bi-directional bandwidth of 25.6 GB/s [40]. The takeaway is that LightWSP ensures correct persist order under LRPO on the cheap.

C. Persist Order Guarantee in Multi-Threaded Program

To recognize the happens-before order dynamically formed by synchronization primitives of multi-threaded program, LightWSP compiler inserts a region boundary, *i.e.,* the PC-checkpointing store, before their instructions such as memory fences and atomic operations. That way, the region ID can be updated at the synchronization point, allowing the sequence of the ID to reflect the happens-before order of threads—even if they touch the same region in a critical section. In particular, LightWSP saves/restores the region ID across context switches to virtualize it. Without virtualizing the region ID, a thread, that was once scheduled out in the middle of a critical section, could not tag its stores with the correct region ID when it is scheduled back in.

Figure 4 exhibits an example of how LightWSP achieves the correct persist order of regions in multi-threaded program. Suppose three threads enter a critical section based on their happens-before order, *i.e., Thread 1* $\rightarrow$ *Thread 2* $\rightarrow$ *Thread 3*. The region ID with which their stores are tagged atomically increases by 1 whenever a boundary is reached (§IV-B), *i.e., N, N + 1, and N + 2* shown in the figure; for the sake of

[5]Thus, each store needs only one bus transaction as usual.

[6]Based on FIFO buffer, store orders are guaranteed.

simplicity, we here ignore the effects of the first boundary assume that all stores target a single MC.

When *Thread 1* leaves the critical section releasing the lock, *i.e.,* it reaches the second boundary in the figure broadcasting the region ID (N), the MC recognizes this with bdry-ACKs exchange and triggers its WPQ to start flushing the store of the thread (❶). Although the WPQ already buffers a store of *Thread 2* (top green WPQ entry in the figure) before receiving *Thread 1*'s boundary, LightWSP enforces the correct persist order of the threads' stores—which follows the original happens-before order—with matching their region ID. That is, the WPQ only flushes *Thread 1*'s stores to PM. After flushig them all, which is notified by flush-ACKs exchanges, the MC updates its flush ID to *N + 1* (§IV-B). Finally, the same process is repeated for *Thread 2* (❷) and *Thread 3* (❸).

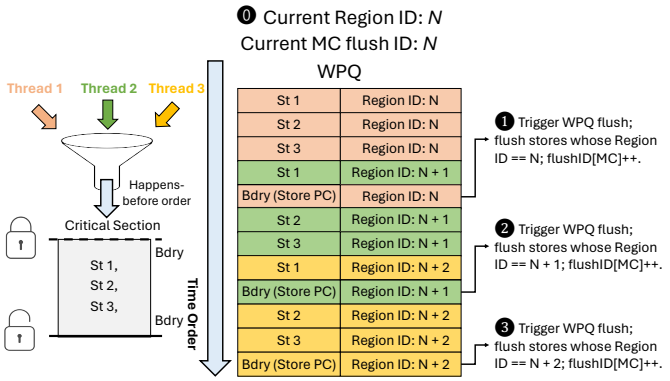

Fig. 4. An example of LightWSP persist order guarantee with multi-threads.

D. Deadlock Resolution

With multithreading or multiprocessing, a deadlock may occur as discussed in §III-C. For deadlock detection, each MC leverages a single bit to indicate whether it contains a region ID (boundary) corresponding to the flush ID. When a WPQ gets full, LightWSP checks if the bit is 0, *i.e.,*, absence of the boundary in the WPQ thus detecting a deadlock—as the WPQ has no room to receive any boundary for which its entries wait. To resolve such a detected deadlock, LightWSP exceptionally lets the WPQ overflow yet with fallback, *i.e.,* the WPQ flushes stores of the region—corresponding to the current flush ID of MC—with them undo-logged. Here, the MC declines other regions' stores but only accepts the stores of the currently persisting region until the WPQ receives the corresponding region ID (boundary). In particular, care must be taken for correct recovery of power failure in case it occurs before the region ID (boundary) is received. In the wake of the power failure, LightWSP first restores the original memory values using their undo log if they ended up being overwritten by the WPQ overflow. Note that according to our experiment, the undo logging does not a significant performance overhead given that the deadlock rarely occurs; see §V-F5.

E. The Size of WPQ

Recall that the region partitioning threshold is determined by WPQ size, and a smaller region size tends to result in more checkpoint stores, leading to performance degradation. Thus, the WPQ size of LightWSP should be sufficiently large but not too much considering the resulting hardware cost. Previous work [35], [60], [101] has deployed 4KB WPQ (64 entries with 64B data granularity), while some designs [53], [100] have utilized 1KB WPQ size (16 entries with 64B data granularity) or 8KB WPQ size (128 entries with 64B data granularity) [3]. However, as revealed by prior work [97], the WPQ in commodity integrated memory controller (iMC) is only 512B (8 entries with 64B data granularity). Hence, to align LightWSP's design with the practical configuration, we set LightWSP's WPQ size to 512B as well with 64 entries (8B granularity). Note that the size of the front-end buffer aligns with the WPQ size, and it is also set to 64 entries.

F. Power Failure Recovery

When power failure occurs, LightWSP conducts a series of actions regarding the flush ID—which indicates the latest unpersisted region (§IV-B). Note that the flush IDs in MCs may be different from each other at the point of power failure—because perhaps only one MC receives all the flush-ACKs by the point, while other MCs are still waiting for the in-flight ACKs. However, those in-flight ACKs are guaranteed to reach their target MCs by leveraging battery power as discussed in §IV-B. Despite power failure, MCs can thus still update their flush ID and start to flush their WPQ.

Specifically, in the event of power failure, LightWSP performs the following steps: (1) In case bdry-ACKs and flush-ACKs are on their way, MCs wait a fixed amount of time, *i.e.,* worst-case ACK communication latency, to ensure that the in-flight ACKs reach their destinations; LightWSP opts for this—rather than retrying the ACKs exchange across MCs—to simplify the remaining steps. (2) MCs retrieve the current flush ID that must be the same across all MCs, whether or not flush ID has just been updated by the failure-atomic ACKs communication in Step 1; recall that such a per MC flush ID is updated only after the MC gets flush-ACKs from all other MCs (§IV-B). (3) To commit only the WPQ entries (stores) of persisted region(s), each MC scans its WPQ entries to check if their tagged region ID matches with the flush ID of the MC, as long as it has received the corresponding bdry-ACKs from all other MCs; the implication is that those entries being flushed belong to persisted region(s). (4) Each MC sends flush-ACKs to other MCs; once the ACKs are fully exchanged among all MCs, the region is committed. (5) Repeat the process from Step 1 to Step 4 until there is no more WPQ entry of any persisted region. (6) All WPQs discard their remaining entries that must be a part of unpersisted region(s).

In particular, LightWSP's recovery process is simple and fast. In the wake of the power failure, LightWSP just resumes the interrupted program from the recovery point[7], *i.e.,*, PC referring to the beginning of the latest unpersisted region, with its input registers restored from their checkpoint.

[7]region ID is restored by reading the flush ID.

G. Buffer Snooping

LightWSP has two different data paths, *i.e.,* the regular data path (caches) and the non-temporal data path. Due to the deep cache hierarchy, cache writeback data virtually always reaches the target MC later than the data placed on the persist path does [51], [92]. However, though chances are extremely low, the cache writeback data may reach the MC earlier, which can break the correct region persist order. Figure 5 shows such an example for which there are two regions; their stores share the same address (*i.e.,* $A = 1$ and $A = 2$). Now, suppose ***A*** is evicted from LLC after several instructions following $A = 2$. Most of the time, the persist path is faster and thus yields an arrival order of ❶→❷→❸, in which case the persist order is enforced correctly. However, in some rare cases, the order might change *e.g.,* because of the memory scheduling delay in the MC or a particular access pattern of program that evicts the cacheline quickly. In this case, other two arrival orders ❶→❸→❷ or ❸→❶→❷ would be possible—though ❶→❷ are always serialized by the FIFO persist path. Under the epoch persistency model [54], [80] that allows in-region ordering yet bans cross-region one, the former case (❶→❸→❷) is allowable, whereas the latter case (❸→❶→❷) is prohibited due the inability to ensure the persist order of the two regions.

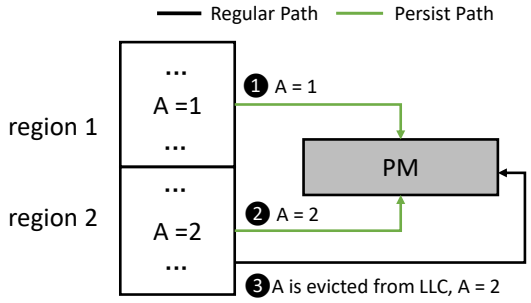

Fig. 5. An example of corrupted persist order.

To prevent such a problem, LightWSP silently drops the dirty cacheline eviction from LLC. However, this may cause a stale-load problem as shown in Figure 6. Suppose that a CPU executes store and load instructions, that have the same address to the cache block ***A***, in sequence. Here, *St A* first completes and reaches LLC after a while (❶). Then, LLC evicts the block ***A*** silently dropping it (❷). After that, *Ld A* encounters an LLC miss so it lets the MC fetch the block (❸). However, this fetch occurs while `St A` is still pending on the persist path. That is why the block brought from PM for the load is not up to date value (❹). Unfortunately, the data of `St A` is arrived at the MC too late (❺), thus causing the stale-load problem.

Taking this into consideration, LightWSP proposes a simple ***buffer-snooping*** to guarantee that a store always reaches MCs earlier than the cacheline eviction. To achieve this, whenever a dirty cacheline eviction happens in L1 cache, LightWSP consults the front-end buffer (via CAM search) to check whether the same data exists therein. If the conflicting entry is found in the buffer (which is said buffer conflict), LightWSP attempts to find another conflict-free victim in the same set

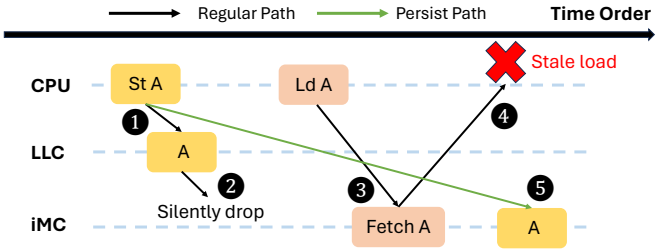

Fig. 6. Stale load problem.

for eviction without unnecessary delay—unlike cWSP [110] which always waits until the entry data persists. To elaborate, if a cacheline in the set also conflicts with any front-buffer entry, LightWSP tries another cacheline and consults the buffer again. This process continues until conflict-free victim is found. In the worst case where the cachelines of the set are all conflicted, the eviction is delayed until the conflicting entry of the buffer is removed—after its data reaches the MC's WPQ with the resulting ACK received by the buffer. In this way, LightWSP can prevent the stale-load problem.

Note that the front-end buffer snooping happens when there is a dirty cacheline eviction in L1 caused by its cache miss. Thus, the search latency (2 cycles; see §V-G2) is hidden by L2 cache access (44 cycles, Table I), and thus never affect the performance. However, the snooping technique impacts L1 cache replacement behaviors and may cause performance degradation. Despite, in our evaluation, due to the deep cache hierarchy, such buffer conflicts are rare and seldom influence performance (see Section V-F3).

H. LLC Load Miss Handling

In the event of an LLC read miss, the requested data might still remain in WPQ. However, LightWSP cannot serve the LLC read miss request from WPQ since the different data granularity, *i.e.,* 8B (WPQ) vs 64B (cacheline). Nevertheless, we found out that WPQ hit rate is quite low (only 0.039 hits per million instructions; see §V-G1). Given this, whenever a memory controller receives a load request, LightWSP simultaneously loads the data from PM and searches WPQ (CAM search). Fortunately, this WPQ search latency can be hidden by the PM load with no extra actions needed. The reason is two-fold: (1) WPQ search (2 cycles; see §V-G2) is faster than PM load, and (2) most of the time, the LLC read request encounters a WPQ miss. While chances are low, the request could lead to a WPQ hit. In this case, LightWSP just drops the previous PM load and reloads the data from PM after the WPQ hit entry is flushed into PM. Note that during the period that a WPQ hit is waiting for its flush, the memory controller can still handle other LLC miss requests in the way mentioned above. That is, both PM load and WPQ search happen in parallel. However, although WPQ hit increases the LLC miss handling time which is thoroughly modeled in our evaluation, the ultra-low WPQ hit rate makes it possible for LightWSP to maintain its high performance.

TABLE I
SYSTEM CONFIGURATION

OS	Ubuntu 18.04 and Linux Kernel 5.4.46
Processor	8-core 4-width OoO processor at 2 GHz, ROB/IQ/SQ/LQ: 224/97/56/72
L1 ICache	32KB/core , 8-way, 64B block, 3 cycles
L1 DCache	64KB/core, 8-way, 64B block, 4 cycles
L2 Cache	16MB shared, 16-way, 64B block, inclusive, 44 cycles
DRAM Cache (LLC)	shared direct-mapped 4GB, DDR4 2400 8x8
PMEM	32GB, read/write=175ns/90ns [100], [102]
Memory Controller	2 MCs, 2 channels/MC 64-entry 8B-granularity WPQ
Persist Path	20ns worst-case latency and 4GB/s bandwidth on-chip network [51], [92]
Front-end Buffer	64-entry FIFO queue

V. EVALUATION

A. Methodology

We implemented LightWSP's compiler techniques described in §IV-A in the LLVM 13.0.1 compiler infrastructure [62]. All evaluated programs including OS and runtime libraries are compiled with the default -O3 flag in addition to enabling our compiler optimizations. We conduct all our experiments on top of the gem5 [10] simulator. It models an Intel Skylake-X 8-core (one hardware thread per core) processor that has two integrated memory controllers (iMC), each of which manages a DRAM cache as an off-chip direct-mapped cache as in Intel PMem's memory mode. Table I shows detailed configurations.

We evaluated LightWSP with both single-threaded (SPEC CPU 2006/2017 [11], [36]) and multi-threaded (Splash3 [89], NPB-CPP [75], STAMP [79] and WHISPER [80]) benchmarks. Our default WPQ size is 64, and the store threshold in each region is 32, *i.e.,* half of the WPQ size. In particular, we set the bandwidth of persist path (non-temporal data path) to 4GB/s as prior work does [92]. We ran these benchmarks using the full-system simulation mode of the gem5 with Linux Kernel 5.4.46 compiled by LightWSP's compiler. Therefore, the OS code is partitioned to a series of recoverable regions with their live-out registers checkpointed and pruned (§IV-A). For SPEC CPU2006/2017, we fast-forwarded 10 billion instructions and simulated the following 5 billion instructions—excluding region boundary (*i.e.,* PC-checkpointing store) and other checkpoint store instructions. On the other hand, we simulated the entire program execution for STAMP, NPB-CPP, WHISPER, and SPLASH3. Lastly, all results are normalized to the unmodified program that does not have region boundary and checkpoint stores.

Our baseline is Intel Optane PMem's memory mode with the original binary. That is, the baseline does not offer persistency or crash consistency, though it takes advantage of the DRAM cache. For performance evaluation, we compare LightWSP with Capri, PPA, and cWSP. In particular, we made some necessary changes to Capri's source code—which we got from the authors—to make it accommodate multiple memory controllers (MCs).

B. Performance Evaluation

Figure 7 shows the slowdown of Capri, PPA, and LightWSP for benchmarks from SPEC CPU2006/2017, STAMP, NPB-CPP, SPLASH3, and WHISPER. Compared with the baseline, Capri, PPA, and LightWSP incur 50.5%, 8.1%, and 9.0% run-time overhead, respectively. As discussed in §II-C, Capri's greater data granularity imposes significant bandwidth pressure on its dedicated persist path, resulting in a non-negligible performance overhead when operating at a 4GB/s bandwidth. It is worth noting that Capri's run-time overhead can be reduced to around 20% with its default setting, *i.e.,* 32GB/s bandwidth, but such high bandwidth might not be practical. In particular, to ensure correct persist order with multiple MCs, Capri must stop the traffic of its separate data path until all the previous region's stores are flushed in PM, leading to non-trivial performance degradation. Notably, LightWSP achieves similar performance with the pure hardware approach PPA which is beneficial from LightWSP's *lazy region-level persist ordering* (LRPO), hiding most of the persistence delay at each region end (§III-B). Even though PPA achieves region-level asynchronous persistence (§II-C), it must wait at each persistence barrier till all the stores of the prior region are persisted.

C. Region-Level Persistence Efficiency

LightWSP proposes *lazy region-level persist ordering* to hide the persistence latency across its recoveral regions. This is one of the reasons for its outstanding performance. In this section, we compare the persistence efficiency of LightWSP and PPA via the following formula.

$$Persistence_{eff}\% = ((T_p - T_{wait})/(T_p)) * 100 \quad (1)$$

T_p is the *Persistence Latency* without using any optimizing techniques. For LightWSP, T_{wait} is the waiting time that it must pay when the WPQ gets full. PPA's T_{wait} is the waiting time at each region boundary (persist barrier). Higher efficiency means that more of persistence latency can be hidden. Figure 8 shows the efficiency results of LightWSP and PPA. Overall, PPA and LightWSP achieve an average efficiency of 89.3% and 99.9%, respectively. Compared with PPA, LightWSP achieves 12.6% higher region-level persistence efficiency, thus more efficiently hiding the persistence delay.

D. PSP vs WSP

To demonstrate the superiority of enabling DRAM as cache, we implement an optimized version of BBB [6] whose performance is close that of Intel eADR [41], [90]. BBB here behaves like an ideal PSP scheme that does not require heavy JIT-checkpointing techniques used in LightPC [63]. Figure 9 compares LightWSP to this ideal PSP scheme for memory-intensive applications and demonstrate the evaluation results. As shown in the figure, LightWSP incurs an average of only 3% performance overhead, while the ideal PSP scheme exhibits a 51.2% performance slowdown on average and up to 260% for `libquantum` of CPU2006 due to the inability to use DRAM as LLC.

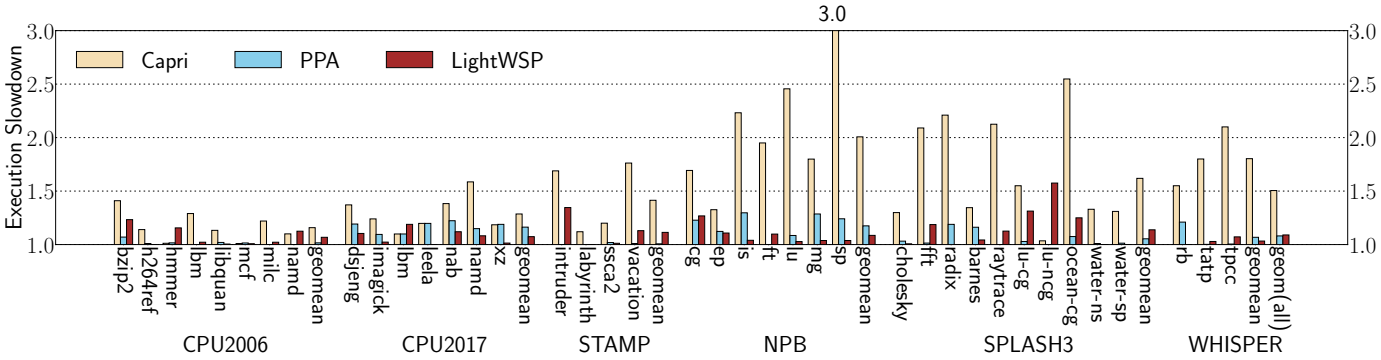

Fig. 7. Slowdown of Capri, PPA and LightWSP compared to the baseline (original binaries running under Intel Optane's memory mode).

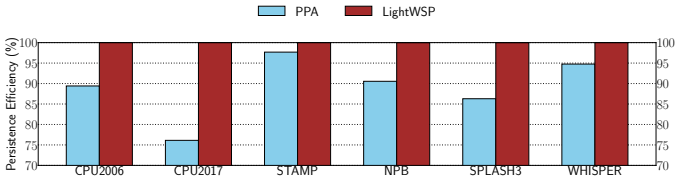

Fig. 8. Region-level persistence efficiency.

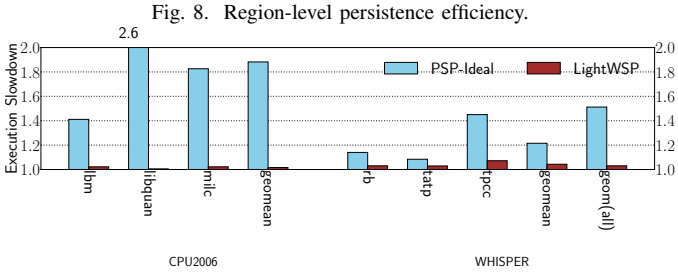

Fig. 9. Slowdown of the ideal PSP scheme and LightWSP for memory-intensive applications.

E. LightWSP vs cWSP

This section compares LightWSP with the state-of-the-art work cWSP. For fair performance comparison, we exclude the NPB benchmark suite as cWSP does not use it. Figure 10 shows average slowdowns of cWSP and LightWSP for the remaining benchmark suites. Although cWSP offers slightly better performance, *i.e.,* an average slowdown of 5.7% compared to LightWSP's 8.5%, cWSP requires intrusive hardware modifications to both cores and MCs—for undo-logging acceleration to mitigate the critical path extension of every write, as discussed in §II-C. In contrast, LightWSP leaves the original microarchitecture of cores/MCs as is yet delivers comparable performance to cWSP.

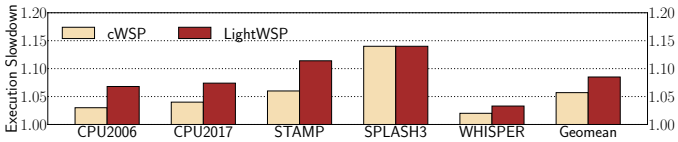

Fig. 10. LightWSP vs cWSP.

F. Sensitivity Analysis

1) WPQ Size: As we discussed before in §IV-E, the size of WPQ can impact the performance. Thus, we conduct a sensitivity study for different WPQ sizes, *i.e.,* 64, 128, 256, corresponding to different region sizes 32, 64, and 128; the size of the front-end buffer remains consistent with the WPQ size. Figure 11 shows the average slowdown of different WPQ size configurations, *i.e.,* 64, 128, and 256, for the same set of benchmark suites used before. Generally, with a smaller WPQ size, LightWSP shows a higher slowdown, while 256-entry WPQ demonstrates the best performance. Note that even with a 256-entry WPQ, its size is only 2 KB, which is significantly smaller than the 8 KB structure (also battery-backed) used in prior work [3]. Given that Intel's eADR [41], [90] battery already supports the JIT-checkpointing of the entire cache hierarchy (around 16MB) upon power failure, it would be possible to enlarge LightWSP 's WPQ for greater scalability targeting those applications with huge write intensiveness.

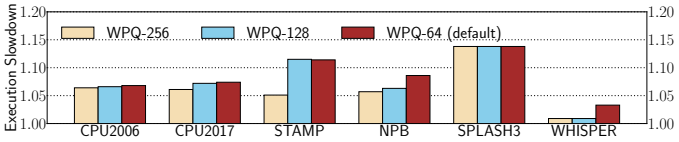

Fig. 11. Slowdown of different WPQ size configurations.

2) Region Size: We also conduct a sensitivity analysis on store thresholds while keeping the same WPQ size, *i.e.,* 64. Figure 12 shows that LightWSP achieves the highest average performance for different benchmark suites, when the store threshold is set to half of WPQ size, *i.e.,* 32. This is because LightWSP achieves a balance between region-level persistence efficiency and live-out register checkpoint overhead.

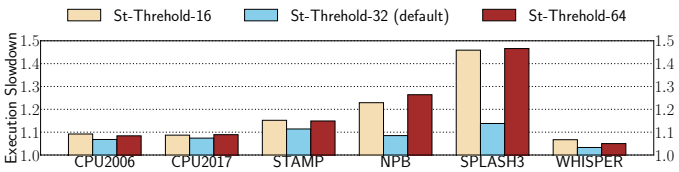

Fig. 12. Slowdown of various store threshold configurations.

3) Cache Eviction Victim Selection: To address the stale load problem, when a data eviction occurs in the L1 cache, LightWSP needs to consult the front-end buffer first. If the buffer has the same data, *i.e.,* buffer conflict, then LightWSP chooses another conflict-free cacheline (victim) to be evicted. This may change cache's LRU replacement behaviors, since some recently used cachelines may be evicted to prevent

TABLE II
CONFLICT RATE

Benchmarks	CPU2006	CPU2017	STAMP	NPB	SPLASH3	WHISPER
Conflict Rate	0‰	0‰	0‰	0.0031‰	0.0012‰	0.00001‰

stale loads. Thus, we try to explore the performance impact on different victim selection policies. To be more specific, we evaluated 3 different victim selection policies, *i.e.,* (1) LightWSP scans all the cachelines (L1 cache is 8-way thus we have 8 victims) to find a conflict-free victim (default); (2) LightWSP only scans only half way through the cacheline list, *i.e.,* 4 cachelines, for victim selection; and (3) LightWSP delays the eviction until the corresponding conflicted buffer entry is flushed, rather than selecting a victim cacheline.

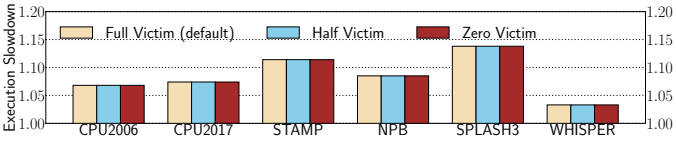

Fig. 13. Slowdown of different victim selection polocies.

Figure 13 shows the performance results of these 3 victim selection policies, *i.e.,* full-victim, half-victim, and zero-victim. The takeaway is that different victim selection policies do not reveal a significant performance difference. That is because the buffer conflict rate is very low as shown in Table II). It highlights that the average conflict rate of all benchmarks is less than 0.01%—though NPB, SPLASH3, and WHISPER have some conflicts.

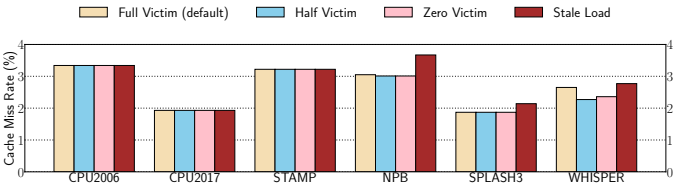

Fig. 14. Cache miss rat with/without buffer snooping.

To better understand the performance of the three victim policies, we measure their cache performance in comparison to that of a special case (stale-load) where L1 cache eviction does not snoop the front buffer—thus suffering the stale load problem. Figure 14 shows that cache miss rates of full-victim, half-victim, zero-victim, and stale-load on average. Overall, the 3 victim policies show similar miss rates while the state-load case shows a higher rate—which is clearly observed in NPB, SPLASH3, and WHISPER; specifically, full-victim, half-victim, and zero-victim policies reduce the cache miss rates of these benchmark suites by up to 12.62%, 17.98%, and 18.05%, respectively.

4) Persist Path Bandwidth: We also explore the impact of the persist path bandwidth on the performance. Figure 15 shows the results of 3 different bandwidths, *i.e.,* 1GB/s, 2GB/s, and 4GB/s. Basically, with the decrease of the persist path bandwidth, LightWSP shows a higher slowdown. That is, the front-end buffer more quickly fills up with lower persist bandwidth, which exerts back pressure on store buffer, possibly leading to pipeline stalls.

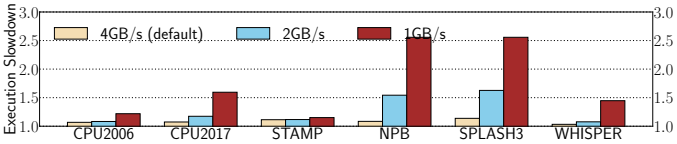

Fig. 15. Slowdown of different bandwidth configurations.

TABLE III
CXL CONFIGURATION

Device	CXL IP	Memory Technology	Max. bandwidth	Latency (read/write)
CXL-I	Hard IP	DDR5-4800	38.4 GB/s	158ns/120ns
CXL-II	Hard IP	DDR4-2400	19.2 GB/s	223ns/139ns
CXL-III	Soft IP	DDR4-3200	25.6 GB/s	348ns/241ns
CXL-PMEM	Simulation	Intel Optane	read:6.6 GB/s write: 2.3 GB/s	245ns/160ns

5) The Number of Threads: This section presents LightWSP's performance with different number of threads (8, 16, 32, and 64). Figure 16 illustrates the results with a fixed WPQ size of 64. As the number of threads increases, the run-time overhead also increases. That is because more threads cause a higher contention on the shared WPQs, resulting in more front-end buffer stalls. With more threads, we also see more WPQ overflow. Despite, the WPQ overflow rate is low—1.9 overflows per 10000 instructions under a 64-thread case. The overflow rate can be reduced by 5.22x by enlarging the WPQ to 256. In addition, for *vacation* application in STAMP, we use low-contention input for 8 threads and high-contention input for other thread numbers, we do not see clear performance degradation when using high-contention input—around 13% run-time overhead at 8-thread and around 14% run-time overhead for other thread numbers.

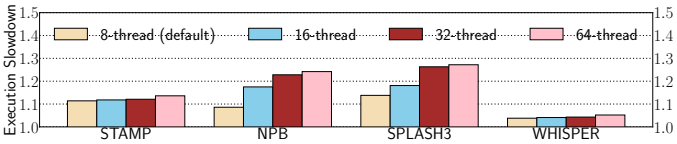

Fig. 16. Slowdown of different thread count configurations.

6) CXL-based LightWSP: Compute Express Link (CXL) emerges as a promising high-performance interconnection technology, offering an alternative approach to using NVM as main memory without going through Intel's integrated memory controller (iMC) as in Optane's memory [24], [74], [78]. Therefore, this section explores LightWSP's performance when CXL is used to connect the persist path to NVDIMM—that can save DRAM contents upon power failure with battery power—or persistent main memory (PMEM). We evaluate four distinct CXL configurations shown in Table III; the first three CXL NVDIMM devices are modeled with parameters drawn from a recent study [94], while the fourth configuration is a CXL-based PMEM device with applying additional 70ns CXL interconnect latency [64]. Figure 17 shows the results, across various configurations. It highlights that LightWSP works well with CXL technologies, causing an average run-time overhead of less than 16%.

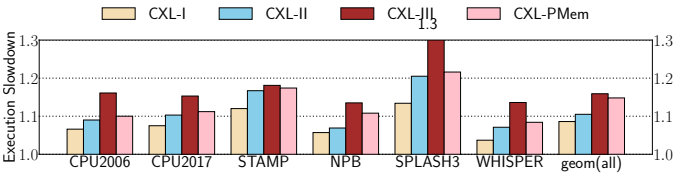

Fig. 17. Slowdown of different CXL configurations.

G. Other Analyses

1) LLC Cache Miss Handling: As mentioned in §IV-H, for LLC load miss, LightWSP simultaneously loads the data from PM and conducts a WPQ search in case the newest data is resident in the WPQ. In case of a WPQ hit, LightWSP has to postpone its PM load until the hit entry is committed to PM. To assess the potential performance impact of this technique, we evaluate the hit rate on the WPQ under our default configuration. As depicted in Figure 18, the WPQ hit rate is very low. Across all the benchmarks evaluated, the average WPQ hit rate is only 0.039 hits per million instructions, indicating a minimal performance impact due to the waiting delay upon a WPQ hit.

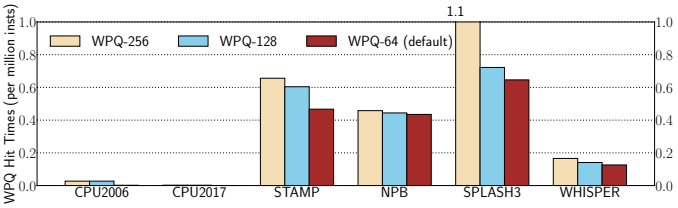

Fig. 18. WPQ hit rate.

2) Front-End Buffer/WPQ Search Overhead: As discussed in IV-G and IV-H, LightWSP consults the front-end buffer and WPQ on L1 cache eviction and LLC miss, respectively. This requires CAM (content-addressable memory) searching. To this end, we use CACTI 7.0 [8] with 22 nm technology to calculate the search delay of both the front-end buffer and the WPQ under 64 entries and an 8-byte granularity. The simulation results highlight the low latency of searching the front-end buffer and the WPQ, *i.e.,* 0.99ns (2 cycles).

3) Instruction Count and Region Statistics: We compare the dynamic instruction count of LightWSP and the baseline. On average, compared with the baseline, LightWSP incurs 7.03% more instructions mainly due to its checkpoint stores. Additionally, LightWSP's average instruction count per region is 91.33 while the average store count per region is 11.29.

4) Hardware Cost Analysis: This section analyzes LightWSP's hardware cost and compares it to PPA and Capri. LightWSP's front-end buffer holds 8-byte entries with a default capacity of 64 entries, totaling 512B. This is fully covered by Intel's 1KB WCB [46]. The WPQ size is also 512B, which aligns with practical configurations in iMC, and does not incur additional hardware cost. MC's hardware overhead is a 2B flush ID per MC. Therefore, in our 8-core configuration with 2 MCs, the hardware cost of LightWSP is near zero, amounting to only 0.5B per core. In contrast, PPA incurs 337B overhead per core due to hardware support for facilitating its store integrity. For Capri, its hardware cost is notably more substantial than PPA, *i.e.,* 54KB per core.

VI. Related Work

Many different schemes have been proposed to provide data persistence and crash consistency. Among these schemes, some only consider partial-system persistence (PSP) which persists only the stores of user-defined regions of code [3], [6], [9], [12], [25], [28], [35], [37], [48], [51], [56], [66], [71]–[73], [76], [77], [82], [83], [85], [86], [88], [92], [99], [100], [103], [104], [113]. However, PSP often requires dedicated interfaces, such as persistent transactions, which are difficult and error-prone, thus burdening programmers. Also, software-based PSP work usually leverages write-ahead-logging (WAL) at each transaction, causing non-trivial performance overhead. In particular, to guarantee the correct persist order, flushes/fences are often introduced which places the logging operations on the critical path thus incurring excessive CPU stalls. While some hardware-based WAL [30], [52], [55], [84], [93] can overlap the logging latency with the execution of subsequent instructions, they still suffer high delay for the synchronization of the persist operations at each transaction end.

LightWSP is not the first to use a separate path to achieve data persistence. Some of work [3], [35], [51], [53], [100], [110] also utilize a separate persist path. However, most of their paths are connected from L1 cache to PM. Their common problem is write amplification causing non-trivial bandwidth pressure on the data path. That is because every 8-byte store generates substantial 64-byte traffic over the path. Instead, LightWSP leverages the non-temporal data path that conveys 8-byte data being stored down to PMEM bypassing the cache hierarchy, thus reducing the bandwidth pressure by 8x.

Finally, previous research has also explored the use of battery-backed buffers to achieve crash consistency [3], [6], [26], [35], [41], [53], [56], [81], [100]. However, some approaches within this spectrum entail substantial battery requirements. For instance, Intel's eADR [41] and Viyojit [56] are typical examples that necessitate a considerable battery size. Besides, other methodologies demand extensive hardware modifications to ensure their correct persist order, such as cache tag extensions (*e.g.,* LAD [35]) or alterations to cache coherence mechanisms, as observed in prior work such as TSOPER [26] and BBB [6]. In contrast, LightWSP only needs minor modification on its MCs (*e.g.,* flush ID), and they demand only a small battery considering the small WPQ size.

VII. Conclusion

This paper presents LightWSP, a lightweight yet performant whole-system persistence (WSP) scheme. LightWSP compiler partitions program into a series of regions whose boundary serves as a recovery point. For efficient persist ordering between the regions, LightWSP proposes *lazy region-level persist ordering* that completely hides their persistence latency. Overall, LightWSP incurs a 9.0% average run-time overhead with near-zero hardware cost, bringing WSP closer to practice.

Acknowledgment

We thank anonymous reviewers. This work was supported by NSF grants 2001124 (CAREER), 2153749, and 2314681.

REFERENCES

[1] "Micron and Intel Announce Update to 3D XPoint Joint Development Program," https://newsroom.intel.com/news-releases/micron-intel-announce-update-3d-xpoint-joint-development-program/, accessed: 2023-03-21.

[2] "Micron NVDIMMs: Persistent Memory Performance," https://www.micron.com/-/media/client/global/documents/products/product-flyer/nvdimm_flyer.pdf, accessed: 2023-03-21.

[3] A. Abulila, I. E. Hajj, M. Jung, and N. S. Kim, "Asap: architecture support for asynchronous persistence," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 306–319.

[4] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, & tools*. Pearson Education India, 2007.

[5] V. A. Alfred, S. L. Monica, and D. U. Jeffrey, *Compilers Principles, Techniques & Tools*. pearson Education, 2007.

[6] M. Alshboul, P. Ramrakhyani, W. Wang, J. Tuck, and Y. Solihin, "Bbb: Simplifying persistent programming using battery-backed buffers," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 111–124.

[7] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson, "Bztree: A high-performance latch-free range index for non-volatile memory," *Proceedings of the VLDB Endowment*, vol. 11, no. 5, pp. 553–565, 2018.

[8] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 1–25, 2017.

[9] A. Bhattacharyya, A. Somashekhar, and J. S. Miguel, "Nvmr: non-volatile memory renaming for intermittent computing," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 1–13.

[10] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[11] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 41–42.

[12] M. Cai, C. C. Coats, and J. Huang, "Hoop: efficient hardware-assisted out-of-place update for non-volatile memory," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 584–596.

[13] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 433–452, 2014.

[14] B. Choi, R. Burns, and P. Huang, "Understanding and dealing with hard faults in persistent memory systems," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 441–457.

[15] J. Choi, J. Choi, J. Hyunwoo, and C. Jung, "Caphammer: Exploiting capacitor vulnerability of energy harvesting systems," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024.

[16] J. Choi, H. Joe, Y. Kim, and C. Jung, "Achieving stagnation-free intermittent computation with boundary-free adaptive execution," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2019, pp. 331–344.

[17] J. Choi, L. Kittinger, Q. Liu, and C. Jung, "Compiler-directed high-performance intermittent computation with power failure immunity," in *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2022, pp. 40–54.

[18] J. Choi, Q. Liu, and C. Jung, "Cospec: Compiler directed speculative intermittent computation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 399–412.

[19] J. Choi, J. Zeng, D. Lee, C. Min, and C. Jung, "Write-light cache for energy harvesting systems," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.

[20] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 105–118, 2011.

[21] A. Conway, A. Gupta, V. Chidambaram, M. Farach-Colton, R. Spillane, A. Tai, and R. Johnson, "{SplinterDB}: closing the bandwidth gap for {NVMe}{Key-Value} stores," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 49–63.

[22] M. Dananjaya, V. Gavrielatos, A. Joshi, and V. Nagarajan, "Lazy release persistency," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1173–1186.

[23] M. De Kruijf and K. Sankaralingam, "Idempotent code generation: Implementation, analysis, and evaluation," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2013, pp. 1–12.

[24] P. Desnoyers, I. Adams, T. Estro, A. Gandhi, G. Kuenning, M. Mesnier, C. Waldspurger, A. Wildani, and E. Zadok, "Persistent memory research in the post-optane era," in *Proceedings of the 1st Workshop on Disruptive Memory Systems*, 2023, pp. 23–30.

[25] B. Di, J. Liu, H. Chen, and D. Li, "Fast, flexible, and comprehensive bug detection for persistent memory programs," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 503–516.

[26] P. Ekemark, Y. Yao, A. Ros, K. Sagonas, and S. Kaxiras, "Tsoper: Efficient coherence-based strict persistency," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 125–138.

[27] G. Fang, J. Choi, and C. Jung, "Hybrid power failure recovery for intermittent computing," in *ACM/IEEE International Conference on Computer-Aided Design (ICCAD)*, 2024.

[28] A. Freij, H. Zhou, and Y. Solihin, "Secpb: Architectures for secure non-volatile memory with battery-backed persist buffers," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 677–690.

[29] K. Genç, M. D. Bond, and G. H. Xu, "Crafty: Efficient, htm-compatible persistent transactions," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 59–74.

[30] E. Giles, K. Doshi, and P. Varman, "Bridging the programming gap between persistent and volatile memory using wrap," in *Proceedings of the ACM International Conference on Computing Frontiers*, 2013, pp. 1–10.

[31] V. Gogte, W. Wang, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Relaxed persist ordering using strand persistency," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 652–665.

[32] H. Gorjiara, *Verifying Correctness of Persistent Memory Programs*. University of California, Irvine, 2022.

[33] H. Gorjiara, W. Luo, A. Lee, G. H. Xu, and B. Demsky, "Checking robustness to weak persistency models," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022, pp. 490–505.

[34] H. Gorjiara, G. H. Xu, and B. Demsky, "Yashme: Detecting persistency races," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 830–845.

[35] S. Gupta, A. Daglis, and B. Falsafi, "Distributed logless atomic durability with persistent memory," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 466–478.

[36] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[37] G. Hodgkins, Y. Xu, S. Swanson, and J. Izraelevitz, "Zhuque: Failure is not an option,it's an exception," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 833–849.

[38] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster, "Nvthreads: Practical persistence for multi-threaded applications," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 468–482.

[39] S.-Y. Huang, J. Zeng, X. Deng, S. Wang, A. Sifat, B. Bharmal, J.-B. Huang, R. Williams, H. Zeng, and C. Jung, "Rtailor: Parameterizing soft error resilience for mixed-criticality real-time systems," in *2023 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2023, pp. 344–357.

[40] Intel, "An Introduction to the Intel® QuickPath Interconnect," https://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html.

[41] Intel, "eADR: New Opportunities for Persistent Memory Applications," https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html.

[42] Intel, "Intel optane persistent memory," https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html, accessed: 2023-03-21.

[43] Intel, "Persistent Memory Learn More Series Part 2," https://www.intel.com/content/www/us/en/developer/articles/training/pmem-learn-more-series-part-2.html.

[44] Intel, "Persistent memory programming," https://pmem.io./.

[45] Intel, "Why Is the Intel Optane Persistent Memory in Memory Mode Not Persistent?" https://www.intel.com/content/www/us/en/support/articles/000055895/memory-and-storage/intel-optane-persistent-memory.html, accessed: 2023-03-21.

[46] Intel, "Intel® 64 and ia-32 architectures optimization reference." https://cdrdv2-public.intel.com/671488/248966-046A-software-optimization-manual.pdf, 2023, accessed: 2023-04-20.

[47] Intel, "Intel 64 and ia-32 architectures software developer's manual," *Volume 3A: System Programming Guide, Part*, vol. 1, p. 64, 2024.

[48] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via justdo logging," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 427–442, 2016.

[49] C. J. Jaeseok Choi, Hyunwoo Joe and J. Choi, "Defending against emi attacks on just-in-time checkpoint for resilient intermittent systems," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024.

[50] J. Jeong, J. Hong, S. Maeng, C. Jung, and Y. Kwon, "Unbounded hardware transactional memory for a hybrid dram/nvm memory system," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 525–538.

[51] J. Jeong and C. Jung, "Pmem-spec: persistent memory speculation (strict persistency can trump relaxed persistency)," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 517–529.

[52] J. Jeong, C. H. Park, J. Huh, and S. Maeng, "Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 520–532.

[53] J. Jeong, J. Zeng, and C. Jung, "Capri: Compiler and architecture support for whole-system persistence," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, 2022, pp. 71–83.

[54] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient persist barriers for multicores," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 660–671.

[55] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "Atom: Atomic durability in non-volatile memory through hardware logging," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 361–372.

[56] R. Kateja, A. Badam, S. Govindan, B. Sharma, and G. Ganger, "Viyojit: Decoupling battery and dram capacities for battery-backed dram," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 613–626, 2017.

[57] H. Kim, J. Zeng, Q. Liu, M. Abdel-Majeed, J. Lee, and C. Jung, "Compiler-directed soft error resilience for lightweight gpu register file protection," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 989–1004.

[58] W.-H. Kim, R. M. Krishnan, X. Fu, S. Kashyap, and C. Min, "Pactree: A high performance persistent range index using pac guidelines," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 424–439.

[59] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-performance transactions for persistent memories," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 399–411.

[60] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated persist ordering," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[61] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating stt-ram as an energy-efficient main memory alternative," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2013, pp. 256–267.

[62] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[63] S. Lee, M. Kwon, G. Park, and M. Jung, "Lightpc: hardware and software co-design for energy-efficient full system persistence," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 289–305.

[64] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal *et al.*, "Pond: Cxl-based memory pooling systems for cloud platforms," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 574–587.

[65] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "Dudetm: Building durable transactions with decoupling for persistent memory," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 329–343, 2017.

[66] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung, "ido: Compiler-directed failure atomicity for nonvolatile memory," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 258–270.

[67] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Clover: Compiler directed lightweight soft error resilience," *ACM Sigplan Notices*, vol. 50, no. 5, pp. 1–10, 2015.

[68] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 228–239.

[69] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Compiler-directed soft error detection and recovery to avoid due and sdc via tail-dmr," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 2, pp. 1–26, 2016.

[70] Q. Liu, C. Jung, D. Lee, and D. Tiwarit, "Low-cost soft error resilience with unified data verification and fine-grained recovery for acoustic sensor based detection," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[71] S. Liu, S. Mahar, B. Ray, and S. Khan, "Pmfuzz: Test case generation for persistent memory programs," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 487–502.

[72] S. Liu, K. Seemakhupt, Y. Wei, T. Wenisch, A. Kolli, and S. Khan, "Cross-failure bug detection in persistent memory programs," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1187–1202.

[73] S. Liu, Y. Wei, J. Zhao, A. Kolli, and S. Khan, "Pmtest: A fast and flexible testing framework for persistent memory programs," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 411–425.

[74] Y. Liu, Y. Ren, M. Liu, H. Li, H. Guo, X. Miao, X. Hu, and H. Chen, "Optimizing file systems on heterogeneous memory by integrating {DRAM} cache with virtual memory management," in *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, 2024, pp. 71–87.

[75] J. Löff, D. Griebler, G. Mencagli, G. Araujo, M. Torquati, M. Danelutto, and L. G. Fernandes, "The nas parallel benchmarks for evaluating c++ parallel programming frameworks on shared-memory architectures," *Future Generation Computer Systems*, vol. 125, pp. 743–757, 2021.

[76] V. J. Marathe, M. I. Seltzer, S. Byan, and T. Harris, "Persistent memcached: Bringing legacy code to byte-addressable persistent memory." in *HotStorage*, 2017.

[77] A. Memaripour and S. Swanson, "Breeze: User-level access to non-volatile main memories for legacy software," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 2018, pp. 413–422.

[78] T. Miemietz, V. Reusch, M. Roitzsch, and H. Härtig, "An nvm performance study towards whole system persistence on server platforms," in *Proceedings of the 1st Workshop on Disruptive Memory Systems*, 2023, pp. 45–51.

[79] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing," in *2008 IEEE International Symposium on Workload Characterization*. IEEE, 2008, pp. 35–46.

[80] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 135–148, 2017.

[81] D. Narayanan and O. Hodson, "Whole-system persistence," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 401–410.

[82] I. Neal, A. Quinn, and B. Kasikci, "Hippocrates: Healing persistent memory bugs without doing any harm," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 401–414.

[83] Y. Ni, J. Zhao, H. Litz, D. Bittman, and E. L. Miller, "Ssp: Eliminating redundant writes in failure-atomic nvrams via shadow sub-paging," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 836–848.

[84] M. A. Ogleari, E. L. Miller, and J. Zhao, "Steal but no force: Efficient hardware undo+ redo logging for persistent memory systems," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 336–349.

[85] S. Pandey, A. K. Kamath, and A. Basu, "Gpm: leveraging persistent memory from a gpu," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 142–156.

[86] S. Pandey, A. K. Kamath, and A. Basu, "Scoped buffered persistency model for gpus," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 688–701.

[87] A. Raad, L. Maranget, and V. Vafeiadis, "Extending intel-x86 consistency and persistency: Formalising the semantics of intel-x86 memory types and non-temporal stores," *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, pp. 1–31, 2022.

[88] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "Thynvm: Enabling software-transparent crash consistency in persistent memory systems," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 672–685.

[89] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2016, pp. 101–111.

[90] S. Scargall and S. Scargall, "Persistent memory architecture," *Programming Persistent Memory: A Comprehensive Guide for Developers*, pp. 11–30, 2020.

[91] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner, "nvm malloc: Memory allocation for nvram." *Adms@ Vldb*, vol. 15, pp. 61–72, 2015.

[92] S. M. Shahri, S. A. V. Ghahani, and A. Kolli, "(almost) fenceless persist ordering," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 539–554.

[93] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for nvm," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 178–190.

[94] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, C. Song, J. Huang, H. Ji, S. Agarwal, J. Lou, I. Jeong *et al.*, "Demystifying cxl memory with genuine cxl-ready systems and devices," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 105–121.

[95] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 91–104, 2011.

[96] T. Wang, S. Sambasivam, Y. Solihin, and J. Tuck, "Hardware supported persistent object address translation," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 800–812.

[97] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao, "Characterizing and modeling non-volatile memory systems," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 496–508.

[98] Y. Wu, B. Min, M. Ismail, W. Xiong, C. Jung, and D. Lee, "{IntOS}: Persistent embedded operating system and language support for multithreaded intermittent computing," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 425–443.

[99] Y. Xu, J. Izraelevitz, and S. Swanson, "Clobber-nvm: log less, re-execute more," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 346–359.

[100] S. Yadalam, N. Shah, X. Yu, and M. Swift, "Asap: A speculative approach to persistence," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 892–907.

[101] F. Yang, Y. Lu, Y. Chen, H. Mao, and J. Shu, "No compromises: Secure nvm with crash consistency, write-efficiency and high-performance," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[102] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory." in *FAST*, vol. 20, 2020, pp. 169–182.

[103] C. Ye, Y. Xu, X. Shen, H. Jin, X. Liao, and Y. Solihin, "Preserving addressability upon gc-triggered data movements on non-volatile memory," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 2, pp. 1–26, 2022.

[104] C. Ye, Y. Xu, X. Shen, Y. Sha, X. Liao, H. Jin, and Y. Solihin, "Specpmt: Speculative logging for resolving crash consistency overhead of persistent memory," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 762–777.

[105] J. Zeng, "Compiler and Architecture Co-design for Reliable Computing," 7 2024. [Online]. Available: https://hammer.purdue.edu/articles/thesis/_b_Compiler_and_Architecture_Co-design_for_Reliable_Computing_b_/26356696

[106] J. Zeng, J. Choi, X. Fu, A. P. Shreepathi, D. Lee, C. Min, and C. Jung, "Replaycache: Enabling volatile cachesfor energy harvesting systems," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 170–182.

[107] J. Zeng, S.-Y. Huang, J. Liu, and C. Jung, "Soft error resilience at near-zero cost," in *Proceedings of the 38th ACM International Conference on Supercomputing*, 2024, pp. 176–187.

[108] J. Zeng, J. Jeong, and C. Jung, "Persistent processor architecture," in *MICRO-56: 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023.

[109] J. Zeng, H. Kim, J. Lee, and C. Jung, "Turnpike: Lightweight soft error resilience for in-order cores," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 654–666.

[110] J. Zeng, T. Zhang, and C. Jung, "Compiler-directed whole-system persistence," in *Proceedings of the 51th Annual International Symposium on Computer Architecture*, 2024.

[111] J. Zha, L. Huang, L. Wu, S.-a. Zheng, and H. Liu, "A consistency mechanism for nvm-based in-memory file systems," in *Proceedings of the ACM International Conference on Computing Frontiers*, 2016, pp. 197–204.

[112] Y. Zhang and C. Jung, "Featherweight soft error resilience for gpus," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 245–262.

[113] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 421–432.

[114] Y. Zhou, J. Zeng, J. Jeong, J. Choi, and C. Jung, "Sweepcache: Intermittence-aware cache on the cheap," in *MICRO-56: 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023.

[115] K. A. Zubair and A. Awad, "Anubis: ultra-low overhead and recovery time for secure non-volatile memories," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 157–168.